PDP-1 COMPUTER

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

CAMBRIDGE 39, MASSACHUSETTS

PDP-30

POSSIBLE

August 27, 1965

POSSIBLE

Introduction

Programming for a digital computer is writing the precise sequence of instructions and data which is required to perform a given computation. The purpose of an assembly program is to facilitate programming by translating a source language, which is convenient for the programmer to use, into a numerical representation or object program which is convenient for the computer hardware to deal with. A symbolic assembly program such as POSSIBLE permits the programmer to use mnemonic symbols to represent instructions, locations, and other quantities with which he may be working. The use of symbolic labels or address tags permit the programmer to refer to instructions or data without actually knowing or caring what specific location in the computer memory they may occupy.

A POSSIBLE source program may be prepared using the standard FIO-DEC Flexowriter with the concise III typeface as given in the Appendix, or using an on-line editing program such as Expensive Typewriter. The source program consists of one or more parts, each with a title, a body, and a start pseudo-instruction. The title is the first non-empty line and is terminated by a carriage return. The body is the storage words, macros, parameter assignments, etc., which make up the substance of the program. The start pseudo instruction denotes the end of the source program. See figure 1.

```
sum
n=100
100/
a,          law tab
            dap b
            dzm s
b,          lac .
            add s
            dac s
            idx b
            sas c
            jmp b
            hlt
tab,        tab+n/
s,          o
c,          lac tab+n
start a
```

Figure 1 - A POSSIBLE SOURCE PROGRAM

POSSIBLE is a two - pass assembler; that is, it normally
processes the source program twice. During the first pass,
it enters all symbol definitions encountered into its symbol
table, which it then uses on Pass 2 to generate the complete
object program. POSSIBLE will either punch a binary tape of
the object program or assemble the program directly onto drum
field 1 during pass 2 of an assembly.

II. POSSIBLE SOURCE LANGUAGE

A. Notation

For clarity the following symbols are assigned to the invisible flexo characters when needed in examples of source program expressions:

carriage return   ↲

tabulation   →|

The abbreviations tab and cr will be used for tabulation and carriage return respectively in format description.

B. Syllables and Expressions

The body of a POSSIBLE source program consists of a sequence of expressions which may be instructions, data, or both. An expression is denoted in the source program by one or more syllables separated by suitable combining operators, and terminated by a tab, cr, slash, comma, or equals. A syllable may be defined as being the smallest element of the programming language which has a numerical or operational value. The following are two of the forms syllables can take:

1. Symbols - A symbol is a string of letters and digits containing at least one letter. Symbols may be of arbitrary length, but are recognized by their first three characters and a test on the existence of any others. Example: sin, sine, and since are all legal symbols, but will be recognized as only two distinct symbols sin and sin_.

2. Integers - An integer is a string of the digits
0, 1, . . . 9. The value of an integer is the
18-bit representation of the integer. Thus, the
largest integer taken as its face value is

777777   in octal or

262143   in decimal.

The value of an integer above these limits is
taken modulo $(2^{18}-1)$. If the integer is immediately
followed by a period (.), then that number is
taken as decimal regardless of the current radix.
Note: Period appearing anywhere within an integer
produces unexpected results.

C. Operators
Syllables may be combined by use of the following operators:

Additive Operators:
1. + or space means additions, modulo $2^{18}-1$ (one's
   complement). A line containing nothing but a
   plus sign or space will not generate a storage
   word.
2. - means subtraction of the syllable. Minus
   signs count out properly; thus, $-+0 = -0 = ---0 = -0$.
   A line containing nothing but a minus sign will
   generate a -0 storage word.

Product Operators:
1. V means logical union (inclusive or)
2. Λ means logical intersection (logical and)
3. ~ means logical inequivalence (exclusive or)
4. x means integer multiply. It performs ones
   complement multiplication - that is, the result
   will be the same as that obtained by repeated
   addition.
5. > means integer division. Division by 0 is
   equivalent to division by 1.
6. < means get remainder of integer division.
   Division by 0 will leave a remainder of 0.

POSSIBLE computes the value of an expression by combining the values of its component syllables.

Operator Priority

Operations of the same priority (on the same line, below) get performed from left to right. Operations of different priorities get performed in the order listed, from top to bottom.

unary - ∨ ∧ ~ × > <
+-
=
( )   These vanish in pairs; priority only
[ ]   important to things inside them.
repeat
→ ↓
,/

The symbols open and close brackets, [,], are used for evaluating an expression before applying other operators; the expression inside is computed before the outside operations are performed. Redundant additive operators are examined and computed from left to right. Redundant product operators are taken as having zero (0) between them and are then computed from left to right. The following examples of symbolic expressions on the left have the value listed on the right. (All numbers are assumed to be octal unless followed by a decimal point.)

EXPRESSION	VALUE
2	2
2+3	5
2-3	777776
2×3	6
2V3	3
2∧3	2
2~3	1
-2~3	777776
5<3	2
13>5	2
7-2V3	4
add 40	400040
claVcma	761200
+-4	777773
--1	1
-+3	777774
++2	2
3××2	0

Other Operators

1. (•) center dot is a null operator that simply
    gets eliminated from an expression whenever
    it is seen within a macro definition. Outside
    a macro definition, a center dot is simply
    ignored. It may be used, for example, with
    the pseudo-instruction character within a macro
    definition to allow a dummy symbol argument to
    appear. See the macro definition part for further
    explanation.

    Example: define   dispatch a,b
                        char 1•a+b
                        terminate

2. (') single quote is a null operator like center
dot but gets eliminated from an expression
whenever it is assembled. This is the most
natural way to concatenate two symbols within
the definition of a macro.

D. Use of Expressions

The meaning of an expression to POSSIBLE is determined
by the context in which it apears in the source program;
the character immediately following the expression usually
indicates its use.

1. Storage Words - An expression followed immediately
by a tab or cr is a storage word.

> Examples: jmp ret ↵
> lac abc →|

The 18-bit number representing the value of the
word is assigned a location in memory; this
location is determined by a location counter in
POSSIBLE. After each word is assigned, the
location counter is advanced by one. Note: A
storage word may be an instruction forming part
of a program, a constant used by the program, or
data.

2. <u>Location Assignment</u> - An expression immediately followed by a <u>slash</u> is a <u>location assignment</u>.

> Examples: 100/
> tab+120/

The current location is set equal to the address portion of the value of the expression.

> Examples: 100/→|sza
> →|jmp 100.

In the source program, the above instructions will cause the instruction sza to be placed at register 100 in the object program, and the instruction jmp 100 will be found in register 101. Note: If, on Pass 1, a location assignment contains any undefined symbol, the definition of address tags is inhibited until the location again becomes definite by means of a defined location assignment. On Pass 2, an undefined symbol will result in an error message (usw). The undefined symbol is taken as zero, and the location remains definite.

3. <u>Symbolic Address Tags</u> - An expression followed
immediately by a <u>comma</u> is an <u>address</u> tag.

    Examples: tab,
               100,
               tab+299,

If the tag is a single undefined symbol, it will
be defined with numerical value equal to the
present value of the location counter. If the
tag is a defined symbol or number, the value
of the expression is compared with the current
location, and a disagreement will cause an
error comment (mdt). If the tag is any other
symbolic expression which contains other than one
undefined symbol, an error printout (ilt) occurs.
Use of a defined symbol as an address tag cannot
change the value of the symbol. Also the current
location cannot be changed by a symbolic address
tag. Using a symbolic address, the preceding
example could be written as:

       100/a2,→|sza
              jmp a2

The programmer should note that location assignments
and symbolic address tags, in themselves, have
no effect on the object program, but rather direct
the process of assembly. Also, he should observe
their inverse character. The location assignment
sets the current location counter to the value
of an expression, while the address tag sets the
value of a symbol equal to the current location.

Hence the sequences:

```
            100/a2,bz,
and         100/ a2,
               bz,
```

each assign 100 as the value of both symbols a2
and bz. A sequence such as

```
            1000/ tab,
            tab+n/
```

is frequently used to reserve a block of
registers for a table of data or computed
results. In the above example, the block
starts at register 1000, is named by the symbol
tab, and contains a number of registers given
by the value of the symbol n.

4. Symbolic Parameter Assignment - A symbol immediately
    followed by an equal sign, an expression, and a
    tab or a cr is a parameter assignment. It assigns
    the symbol to the left of the equal sign a
    numerical value given by the expression to the
    right, if the latter is defined. If the expression
    is undefined, no action is taken.

```
            Examples:  n=100
                       sna=sza 1
                       cai=claVcli
                       t=t+t
```

Parameter assignment may be used to set table
sizes, define new operation codes, or prepare a
set of instructions for an interpretive program.
Note: If equal sign (=) is immediately preceded
by a number, POSSIBLE complains. An expression
such as $z3+k9=y8$ defines the symbol $k9$ with the
value of symbol $y8$ and generates a storage word
$z3+k9$; it does not cause the symbol $y8$ to be
evaluated as $z3+k9$. The expression $k9=z3=y8$, if
$y8$ is defined, assigns both symbol $k9$ and symbol
$k9$ and symbol $z3$ the numerical value of symbol $y8$.

E. Comments

The character slash, /, when not preceded by an expression,
denotes the beginning of a comment. Characters following it
are ignored by POSSIBLE until the next carriage return.

F. Current Location Counter

The POSSIBLE location counter records the assignment
location for each word in the object program. It is set
to 4 at the beginning of each pass, and counts upward
modulo memory size. As was explained earlier, the location
counter may be set to any value by a location assignment
expression. The character period (.) when not preceded by
a number, is a special syllable whose value is equal to the
current location. Hence,

```
        sza
        jmp .-1
```

is an alternate way of writing

```
        a2,→|sza
             jmp a2
```

G. Radix 50 Sqoze Codes

The character double quote (") can be used to generate
a radix 50 sqoze code for the first three characters of the
preceding symbol. If there are more than three characters in
the syllable, bit 1 is set to 1.

H. Pseudo-Instructions

Normally an assembly program produces one machine language instruction for each instruction of the source program. However, some lines in the source program, known as pseudo-instructions, are directions to the assembler and do not directly produce instructions in the object program. These instructions govern the way in which subsequent information in the source program is processed.

A pseudo-instruction is a string of at least four letters and digits, in which at least one of the characters is a letter. The string is terminated by an operator. A pseudo-instruction may always be shortened to four characters.

The pseudo-instructions of POSSIBLE are described below:

1. End of Source Program - The pseudo-instruction start denotes the end of the source language program. The expression following start gives the address of the instruction in the object program which is to be executed first. POSSIBLE stores this address and if a binary tape of the object program is produced, POSSIBLE will include an appropriate start block in the binary program tape.

   Example: start beg+2

   This line will terminate scanning of the source program and if the object program is being punched then the word jmp beg+2 will be punched for the start block of the binary tape. When the binary tape is read into the PDP-1 in read-in mode control will go to register beg+2 after the start block is read.

2. <u>Radix Control</u> - The pseudo-instructions <u>octal</u>,
   <u>decimal</u>, and <u>radix</u> control the current numeric
   base for evaluation of integer syllables. The
   pseudo-instruction <u>octal</u> located anywhere in the
   source program indicates all integers following
   it [unless specifically denoted as decimal by a
   period (.)] are interpreted as octal numbers
   until a next appearance of the pseudo-instruction
   <u>decimal</u> or <u>radix</u>. The word <u>decimal</u> indicates
   all integers following it are interpreted as
   decimal numbers until the next appearance of the
   pseudo-instruction <u>octal</u> or <u>radix</u>. The pseudo-instruction
   <u>radix</u> takes any expression following (until the
   next <u>tab</u> or <u>cr</u>) as the new radix. Numbers used
   as the argument of <u>radix</u> are assumed to be decimal.
   If the radix is not defined, it is taken as octal.
   Note: The largest integer taken at its face value
   is 262143_{10} or 777777_8.

3. <u>Storage of Character Codes</u> - The pseudo-instructions
   <u>character</u>, <u>flexo</u>, and <u>text</u> are provided to the
   programmer as a convenient means of storing
   character codes for printout by his program, or
   for comparison against alphanumeric data accepted
   by his program. For reference, the six-bit codes
   for the concise III character set used with the
   PDP-1 are included in the Appendix of this memorandum.

(a) The pseudo-instruction <u>character</u> is used
to place a character code in the left
(bits 0-5), middle (bits 6-11), or right
(bits 12-17) portion of the word. The
word <u>character</u> is followed by space, then
by a <u>r</u>, <u>m</u>, or <u>l</u> according to the position
desired, and then the character whose code
is **wanted**.

		VALUE
Examples:	char ra	000061
	char mb	006200
	char lc	630000

The above are <u>pseudo-instructions syllables</u>,
and may be used in the same manner as symbols
or integers in forming expressions.

		VALUE
Examples:	-char ra	777716

(b) The pseudo-instruction <u>flexo</u> is used to
compile three character codes into one
eighteen bit word.

		VALUE
Example:	flex abc	616263

This is equivalent to:

char ra + char mb + char lc

(c) The pseudo-instruction <u>text</u> is used to
assemble a long string of characters by groups
of three into successive words in the object
program. The string to be assembled is enclosed
between two appearance of the same character
and is preceded by the word <u>text</u>. The character
selected as a delimiter cannot appear in the
string itself.

Examples: VALUE
    1. text .message.   446522
                                          226167
                                          650000

This is equivalent to:
    flexo mes
    flexo sag
    char le

    2. text /this is printed/ 237071
                                   22C071
                                   220047
                                   507145
                                   236564

which is equivalent to:
    flexo thi
    flexo s i
    flexo s p
    flexo rin
    flexo ted

Any expression before the text is added to
the first word of the text; any expression
immediately following the range of a text
is added into the last word of the text
unless the number of characters in the
range of the text, modulo 3, is zero. In
this case, the expression is assembled into
a word of its own. This is useful, for
example, when one wishes to type an expression
in red and so needs to introduce red and black
shifts into the text. Thus,

    350000+text . this gets printed in red. +34
assembles a red shift and a black shift into
the text.

4. Repeat Pseudo-Instruction - The repeat pseudo-instruction provides a convenient way of placing a sequence of similar expressions in a block of the object program. The pseudo-instruction repeat is followed by a symbolic expression, a comma (operator with priority higher than tab may be used), and the range of the repeat. The latter contains all the material from the comma to (and including) the next carriage return. This pseudo-instruction causes POSSIBLE to scan and assemble the range a number of times equal to the value of the expression immediately following repeat. The symbolic expression must be defined when the repeat is encountered during pass 1 and cannot have a value greater than 400000_8; if it is negative or zero, the range of the repeat is ignored. The range of the repeat can be storage words, parameter assignments, macro calls (if not containing carriage return in an argument), other repeats, or anything else. If repeat is used in the range of a repeat, both repeats will end on the same carriage return. However, open and close brackets may be used to enclose the range of the inner repeat and thus, allow cr to appear within the range. Arithmetic brackets may not be used in the range of a repeat unless the entire range is enclosed by brackets; the number of open and closed brackets must be the same. Repeat may be used in macros; dummy arguments may appear either in the range or the count of the repeat, or both.

Example: 1. repeat 3, ril 6s →| tyo
              will assemble the following
              instructions:
              ril 6s
              tyo
              ril 6s
              tyo
              ril 6s
              tyo

          2. repeat 2, [5 →| repeat 2,3
              ] →| 1
              will assemble the following:
              5
              3
              3
              1
              5
              3
              3
              1

5. Conditional Assembly - It is often useful, particularly
    in macro instructions, to be able to test the value
    of an expression, and to make part of the assembly
    dependent on the result of this test. For this
    purpose the pseudo-instruction whenever is provided.
    Following the pseudo-instruction there is a symbolic
    expression, a comma, and the range. The latter
    contains all the material from the comma to (and
    including) the next carriage return. This pseudo-
    instruction causes POSSIBLE to evaluate the symbolic
    expression following the repeat and if its value
    is zero, the range will be assembled once. The
    symbolic expression must be defined when the whenever
    is encounted during pass 1; an undefined symbol is
    taken as zero. The range of the whenever can be
    storage words, parameter assignments, macro calls
    (if not containing carriage return in an argument),
    repeat pseudo-instructions, or anything else.

If <u>repeat</u> is used in the range, both the <u>whenever</u> and <u>repeat</u> pseudo-instructions will end on the same carriage return.

> Example: whenever n, lac tab
> If n=0, this will assemble the storage word, lac tab, once. Thus, this instruction would be equivalent to
>> repeat 1, lac tab

6. <u>Special Tape Format</u> - For fabricating special tape formats or punching start blocks without stopping the assembly, the pseudo-instruction <u>word</u> is provided. It takes one argument ended by a tab or carriage return; this argument is punched directly onto the object program tape during pass 2. The location counter is not affected by this pseudo-instruction.

7. <u>Informative Printouts</u> - The pseudo-instructions <u>printx</u> and <u>value</u> can be used to generate informative printouts during an assembly. <u>Printx</u> takes an argument whose format is exactly like the pseudo-instruction <u>text</u>. During an assembly, POSSIBLE will print out this argument on line. The pseudo-instruction <u>value</u> takes an argument which is an arithmetic expression and prints out its octal value during an assembly.

I. Automatic Storage Assignment

Several features have been provided in the POSSIBLE
assembly program which automatically assign storage locations
for the constants used by a program and the variables and
tables manipulated by the program. These features reduce
the amount of typing required to prepare a complete source
language program, simplify editing, and make the source
program typescript more readable.

1. Constants - An expression enclosed in parentheses
    is a constant syllable and may appear as a syllable
    in storage words and parameter assignment. POSSIBLE
    will compute the value of the expression enclosed
    and place it in a constants area of the object
    program as explained below. The value of a
    constant syllable is the address where the
    enclosed word is placed by POSSIBLE. The location
    at which constant words are placed is determined
    by the next appearance of pseudo-instruction
    constants following the constant syllable.
    When the pseudo-instruction constants is scanned
    by POSSIBLE, the constant expressions assembled
    since the last use of the pseudo-instruction
    constants, or since the beginning of the program,
    are placed in the object program starting at
    the current location. Constant words having the
    same numerical value are entered only once. The
    current location is advanced to an address
    somewhat beyond the register in which the last
    constant is placed, leaving a small gap of unused
    registers between the constants area and any
    following portion of the program. This gap arises
    because POSSIBLE reserves one location for each
    symbolic and each unique numeric constant during
    the first pass but may be able to do some combination
    on pass 2.

Note: The close parenthesis may be omitted from
constant syllables immediately followed by one
of the terminating characters comma, tab, close
bracket, or cr. Recursive use of constant syllables
is permitted; that is, a constant syllable may
appear within an expression forming a new constant
syllable.

Example: Thus, the sequence
lac (lac tab
dap .+1
a2, 0
constants
is equivalent to
lac abc
dap .+1
a2, 0
abc, lac tab.

2. Variables - A symbol typed with a bar over at
least one of its characters at any appearance
in the source program is a variable. All symbols
identified as variables become defined on the
subsequent appearance of the pseudo-instruction
variables. The pseudo-instruction variables
must follow all defining appearance of variables.
The variables are assigned to sequential locations
starting at the location of the pseudo-instruction
variables. Their initial contents is indefinite.

Example: The sequence
              lac a2,
              add bz,
              dac a2,
              bz, 0
              a2, 0
is equivalent to
              lac ā2
              add b̄z
              dac ā2
              variables
except that the contents of registers
a2 and bz of the object program will
be zero in the first case, and unknown
in the second.

3. Tables - Blocks of registers may be reserved for
    tables by means of the dimension pseudo-instruction.

       Example: dimension x(n), y(m), z(m+n),

This string reserves three blocks of lengths given
by the values of the expressions n, m, and m+n.
The first address of each block is assigned as
the value of the symbols x, y, and z. The
reserved blocks are placed at the location in the
object program specified by the variables pseudo-
instruction. The initial contents of the reserved
blocks is indefinite in the object program. The
following rules apply:

(a) Expressions given as lengths of blocks in
     a dimension pseudo-instruction must be
     definite when scanned on the first pass.

(b) The symbols assigned to blocks by a <u>dimension</u>
     statement must be previously undefined.

The use of <u>dimension</u>, <u>variables</u>, and <u>constants</u> in a complete
POSSIBLE source program is illustrated in figure 2. This
program will produce exactly the same object program as the
introductory example on page 2 except that the initial contents
of register s̲ is zero in the earlier version and undefined here.

```
sum
n=100
dimension tab(n)
100/
a,  law tab
    dap b
    dzm s̄
b,  lac .
    adm s
    idx b
    sas (lac tab+n
    jmp b
    hlt
variables
constant
start a
```

<u>Figure 2</u>

J. Macro Instructions

Often certain character sequences appear several times throughout a program in almost identical form. The following example illustrates such a repeated sequence.

```
lac a
add b
dac c
lac d
add 1
dac f
```

The sequence:

```
lac x
add y
dac z
```

is the model upon which the repeated sequence is based. This model can be defined as a macro instruction and given a name. The characters x, y, and z are called dummy arguments, and are identified as such by being listed immediately following the macro name when the macro instruction is defined. Other characters, called arguments, are substituted for the dummy arguments each time the mode is used. The appearance of a macro-instruction name in the source program is referred to as a call. The arguments are listed immediately following the macro name when the macro instruction is called. When a macro instruction is called, POSSIBLE reads out the characters which form the macro-instruction definition, substitutes the characters of the arguments for the dummy arguments and assembles the resulting characters into the object program.

Examples: define absolute

                  spa

                  cma

                  terminate

        define move a, b

                  lac a

                  dac b

                  terminate

Note: If an argument expression is omitted, then the null string (no character) is inserted for that dummy argument. This differs from MACRO; the arguments are not evaluated but are substituted as text strings.

1. Defining a Macro-Instruction - A macro-instruction consist of four parts; the pseudo-instruction define, the macro-instruction name and dummy symbol list, the body, and the pseudo-instruction terminate. Each part is followed by at least one tabulation or carriage return. The macro instruction name has the same form as a pseudo-instruction - a string of letters and digits of which at least one of the first six characters is a letter. The name is terminated by a space, or if there is no dummy symbol list by a cr or tab. The first six characters of a macro instruction name must distinguish that name from all other macro names and from all pseudo-instructions. If a name is three or less characters long, it must be spelled out in full but if it is longer, it may be abbreviated, like a pseudo-instruction, to four characters. The dummy symbol list consists of up to 13_8 or 11_{10} distinct dummy symbols, separated from each other by commas, and from the macro name by a space. Since dummy symbols have no meaning outside of a macro definition, the same symbols may be used in many definitions without harm.

The body of a macro definition is an arbitrary
sequence of expressions in which any dummy symbol
list may appear as a syllable. All the pseudo-
instructions can be used within the body of a macro
definition. This includes the pseudo-instruction
define which must have its own terminate
pseudo-instruction. The body may also contain
macro calls, including calls to the macro itself.

Example: The definition and use of a
macro instruction is illustrated
by a program to store zeros
in a block of register. This
program can be assigned the
name clear by the definition:

```
Define          clear a,n
                law a
                dap .+1
                dzm
                idx .-1
                sas (dzm a+n
                jmp .-3
                terminate
```

When the line

```
                clear tab,100
```

appears later in the source
program, the instruction sequence

```
                law tab
                dap .+1
                dzm
                idx .-1
                sas (dzm tab+100
                jmp .-3
```

is inserted into the object program.
The resulting sequence will clear
a hundred registers starting with
register tab.

K. __Format__

POSSIBLE has few requirements on format. The user should
be aware of the following:

1. Carriage returns and tabs are equivalent except
in the title, in the range of a repeat, in a
comment, and after __start__. Extra tabs or carriage
returns are ignored.

2. Backspace, ⊃, ↑, →, ?, ↲, |, red, black, and
unused characters of the flexo code are illegal
except in arguments of flexo code pseudo-instructions,
titles, and comments.

3. Stop codes are ignored except in arguments of
flexo code pseudo-instructions. Apostrophes
are similarly ignored when not in macro calls
or definitions.

4. Deleted characters are always ignored.

Many programmers have found that adherence to a fairly
rigid format is of help in writing and correcting programs.
The following suggestion have been found useful in this
respect:

1. Place address tags at the left margin, and run
instructions vertically down the page indented
one tab stop from the left margin.

2. Use only a single carriage return between instructions,
except where there is a logical break in the flow
of the program. Then put in an extra carriage
return.

3. Forget that you ever learned to count higher than three; let POSSIBLE count for you. Do not say dac .+16; use an address tag. This will save grief when corrections are required.

4. Have the typescript handy when assembling or debugging a program, and note corrections in pencil thereon as soon as you find them.

5. As macro instructions must be defined before they are used, put these definitions at the beginning of the program.

6. If the pseudo-instructions constants and variables are used, they should be placed before the start at the end of the program.

III. POSSIBLE ASSEMBLY

POSSIBLE is a two pass assembler; that is, it normally processes the source program twice. During the first pass, it enters all symbol definitions encountered into its symbol table, which it then uses on pass 2 to generate the complete object program. POSSIBLE was written for time-sharing mode; its commands are typed through the console. The assembler may be used out-of-time sharing also by typing commands.

POSSIBLE may get the source program directly from expensive typewriter's text buffer or from the paper tape reader. A resulting program is either assembled on (drum) field 1 or punched out as a binary tape. [It is also possible to do an assembly without any output just to check for errors.] The symbol table formed during the assembly may be typed or punched out in numeric or alphabetic order. Also the area used for storing constants and variables may be typed out.

It is possible to fabricate tapes with special formats such as a jump block replacing the input routine or several titles and input routines on one tape.

The programmer may leave POSSIBLE to go to ID, the debugger program, or to the Editor, to correct errors in the source program found during assembly.

A. Possible Assembly Control Characters

The control of the POSSIBLE assembly procedure is by typed-in commands. The following tables indicate the commands that are available and what they mean:

COMMANDS	-29-	MEANING

INPUT SOURCE:

e	expensive typewriter text buffer source
o	off-line source (reader)

OUTPUT MEDIUM:

d	drum assembly
t	tape assembly
w	without output

COMMAND MODIFIERS:

g	get
x	cancel (exchange)

SPECIAL FORMAT:

(g,x) i	input routine
(g,x) j	jump block
(g,x) l	label (title)

ASSEMBLY CONTROL

s	start new pass (also used to suppress punching after error printout)
c	continue pass (also used to continue punching after error printout)
1	pass 1
2	pass 2
f	forget everything (initialize symbol table)

IO EQUIPMENT CONTROL

(g,x) r	reader (initialize reader buffer non ts mode
(g,x) p	punch

SYMBOLS

a	alphabetic symbols
n	numeric symbols
k	konstants areasaand variables areas

EXIT

b	back to ID or ADM RT
m	meliorate source program (back to ET)

SENSE SWITCH USAGE

POSSIBLE uses sense switches during an assembly to provide the following special features:

SENSE SWITCH	USE
1	type-out characters dispatched on
4	continue assembly without stopping after any error printouts.
5	listen for input from typewriter
6	suppress checking for parity error.

The symbol package for POSSIBLE also uses sense switches to indicate additional information.

SENSE SWITCH	USE
1	suppress punching and typing
2	down - punch out symbols
	up - type out symbols
3	down - input format
	ex. tab=105
	up-listing format
	ex. tab →| 105

If POSSIBLE is entered from Expensive Typewriter by the command N (nightmare version of POSSIBLE) or M (merging version of POSSIBLE), POSSIBLE is initially set up to accept source from expensive typewriter and to place the resulting binary object program onto (pseudo) drum field 1. Otherwise, POSSIBLE will expect input source from the paper tape reader (off-line) and will punch a paper tape of the object program. These conditions may be altered by using the appropriate commands, "e" (for source from expensive typewriter's text buffer), "o" (off-line reader used for source program), "d" (assemble onto drum field 1), "t" (assemble onto paper tape), and "w" (without output; just check for errors).

If the input source program is expected from the reader, POSSIBLE will automatically assign the reader at the start of each pass. If the reader is busy, an "r" is type out. Typing "s" gets the reader if it is no longer busy and starts the pass again. Typing "gr" is used to initialize the reader buffer. If on pass 2, a tape is to be punched and POSSIBLE is unable to get the punch assignment, a "p" will be typed. Typing "s" will get the punch if it is no longer busy and will start the pass again.

If a binary object program is punched during pass 2 of an assembly, it will contain a title in readable characters, consisting of the visible characters in the title up to but excluding a center dot. Next will be punched an input routine, which is a loader that reads in the rest of the tape, and which may itself be read in by the PDP-1 read-in mode. The binary output from the body of the source program is punched in blocks of up to 100 registers. The end of the binary tape is denoted by a start block, which is produced by typing "s" after pass 2 is completed. The start block causes the input routine to transfer at once to the address specified by the pseudo-instruction start. The argument of the start has the value of the address to which control is to be transferred.

For fabricating special tape format, the control characters "i", "j", and "l" may be used. Typing "gi" will cause an input routine to be punched when the next tape is assembled during pass 2. Typing "xi" suppresses the punching of an input routine when the next tape is assembled. Typing "gj" causes a jump block (jmp 7753-input routine) to be punched when the next tape is assembly during pass 2. Typing "xj" cancels an "gj" command. Typing "gl" causes a title in readable format to be punched when the next tape is assembled during pass 2. Typing "xl" causes no title to be punched during the assembly.

Normally each pass is initiated by typing an "s" (start pass). To process each additional tape type "c" (continue pass). During Pass 2, if an error is encountered, typing "s" will continue the assembly but suppress punching a binary tape; typing a "c" will continue the pass and continue the punching of the tape. To alter the normal assembly process, the commands "1", "2", may be used. Typing "1" will cause the next "s" to initiate pass 2; typing "2" cause the next "s" to initiate pass 2. To initialize the symbol table so that assembly done previously is forgotten, "f" is typed.

It is sometimes useful to type in on-line short programs, symbol definitions, etc. This may be done by having sense switch 5 up when "s" or "c" is typed. Instead of reading tape, POSSIBLE will listen to the typewriter until sense switch 5 is turned off. Sense switch 1 may also be used during the assembly. If it is up, POSSIBLE will type out each character which is dispatched on. Note that this is not equivalent to typing out every character processed. For example, arguments of the pseudo-instruction character will not be typed and some characters may be typed more than once. This sense switch usage provides aid in debugging especially complicated macro definitions. Sense Switch 4 up allows the assembly to continue without stopping after any error printouts. The punching of a binary program is not affected if the switch is up. Sense switch 6 will suppress the check for parity errors in the input source program.

A copy of the symbols and their respective values may be obtained after the assembly process. The commands "a" (alphabetic symbols), "n" (numeric symbols), and "k" (konstants areas and variables areas), and sense switch 2 which is up to type symbols, down to punch; sense switch 3 which is up for input format, down for listing formats; and sense switch 1 for suppressing both punching and typing, are used to request the desired form.

B. Normal Assembly Procedure

1. To begin pass 1 on the source program, type "s".
    POSSIBLE will stop shortly after encountering
    the cr after the pseudo-instruction start at the
    end of the tape.

2. To process each additional tape after the first,
    type "c".

3. Begin pass 2 by typing "s". At this point, if
    POSSIBLE is to produce a binary tape, it punches
    some blank tape, the title at the beginning of
    the tape in readable form, a binary input routine
    in read-in mode, and then begins to punch the
    binary version of the program in blocks of 100
    words (or less). POSSIBLE, as on pass 1, will
    stop after encountering the start at the end of
    the tape.

4. To process each additional tape during pass 2,
    type "c".

5. If a binary tape is being produced by the assembly,
    an "s" should be typed to punch the start block
    at the end of the tape.

This completes the assembly process.

C. Error Comments During A Possible Assembly
    Upon detecting an error, POSSIBLE will print out a line
in the following format;
        aaa bbbb ccc dddd eee
where aaa is the three letter code indicating the error, bbbb
is the octal address at which the error occurred, ccc is the
symbolic address at which the error occurred, dddd is the name
of the last pseudo-instruction encountered. In the case of
an error caused by a symbol, eee will be that symbol. Following
is the list of error indications in POSSIBLE:

ERROR	MEANING
nca	NO CONSTANTS AREA
	The pseudo-op constants is needed.

ilf	ILLEGAL FORMAT

ilt	ILLEGAL TAG
	Tag which is not a single symbol is
	not equal to current location.
	ex. foo+10, $\neq$ current location

mdt	MULTIPLY DEFINED TAG
	Tag consisting of a single defined symbol
	is not equal to current location. Symbol
	is not redefined.

usw	UNDEFINED SYMBOL
	A symbol which has not been defined in
	program is encountered. Symbol is given
	the value of zero if assembly is continued.

cld	CONSTANTS LOCATION DIFFERENT
	The constant pseudo-op appears in different
	location on pass 2. No recovery can be made.

vld	VARIABLES LOCATION DIFFERENT
	Same as cld but for variables. Often
	possible to recover by ignoring this.

ild	ILLEGAL DEFINITION
	Program attempts to redefine pseudo-op
	or previously defined symbol. Redefines
	pseudo-op or symbol if assembly continued.

sce	STORAGE CAPACITY EXCEEDED
	Storage of macro definitions, macro arguments,
	repeat ranges, numerical constants (pass 1),
	unique constants (pass 2), symbols, or macro
	names has been filled. No recovery can be made.

ERROR	MEANING
pce	PUSH DOWN CAPACITY EXCEEDED

Macro, repeat, or constant nesting is
too deep or too complicated arithmetic
statements are used. No recovery can
be made.

tmc   TOO MANY CONSTANTS AND VARIABLES PSEUDO-OPS

Total numer of constants and variables
pseudo-ops is 20_8. No recovery can be made.

mdd   MULTIPLY DEFINED DIMENSION

Symbol representing first location in
dimension of array is already defined. The
old symbol definition is retained if
assembly is continued.

tmt   TOO MANY TERMINATE PSEUDO-OPS

There exist more terminate instructions
than define instructions. The terminate
is ignored if assembly is continued.

ids   ILLEGAL DIMENSION SIZE

Specified dimension size is negative.
Dimension size is set to zero if assembly
continued.

ich   ILLEGAL CHARACTER

Input source has an illegal flexo code
or character. Number typed is the illegal
character; if the number is in the 400's
it is an upper case character. Continuing
assembly will ignore the character.

APPENDIX

POSSIBLE SYMBOL TABLE

BASIC INSTRUCTIONS		IN-OUT TRANSFER GROUP		SHIFT/ROTATE GROUP	
add	400000	cbs	720056	ral	661000
adm	360000	cks	720033	rar	671000
and	020000	dba	720061	rcl	663000
cal	160000	dcc	720062	rcr	673000
dac	240000	dia	720060	ril	662000
dap	260000	dpy	730007	rir	672000
dio	320000	dra	720063	sal	665000
dip	300000	eem	724074	sar	675000
div	560000	esm	720055	scl	667000
dzm	340000	ioh	730000	scr	677000
idx	440000	iot	720000	sft	660000
idz	120000	lem	720074	sil	666000
ior	040000	lsm	720054	sir	676000
iot	720000	ppa	730005		
isp	460000	ppb	730006	MISCELLANEOUS	
jda	170000	rpa	730001		
jdp	140000	rpb	730002	clo	651600
jmp	600000	rrb	720030	i	010000
jsp	620000	tyi	720004	1s	1
lac	200000	tyo	730003	2s	3
law	700000			3s	7
lio	220000	OPERATE GROUP		4s	17
mul	540000			5s	37
opr	760000	cla	760200	6s	77
sad	500000	clc	761200	7s	177
sas	520000	clf	760000	8s	377
sft	660000	cli	764000	9s	777
skp	640000	cma	761000		
sub	420000	hlt	760400	TIME SHARING INSTRUCTIONS	
xct	100000	lai	760040		
xor	060000	lap	760300	arq	722277
		lat	762200	asc	720051
SKIP GROUP		lia	760020	bpt	722177
		nop	760000	cac	720053
clo	651600	opr	760000	ckn	720027
skp	640000	stf	760010	dsc	720050
sma	640400	swp	760060	dsm	722377
sni	644000	xx	760400	isb	720052
spa	640200			lea	724677
spi	642000			lei	724577
spq	650500			nmf	725477
sza	640100			nmn	725377
szf	640000			rbt	720237
szm	640500			rer	724777
szo	641000			wat	722477
szs	640000				

APPENDIX (cont.)

ALPHANUMERIC CODES

Character		FIO-DEC Code	Concise Code	Character		FIO-DEC Code	Concise Code
a	A	61	61	0	→	20	20
b	B	62	62	1	"	01	01
c	C	263	63	2	ı	203	03
d	D	64	64	3	~	04	04
e	E	265	65	4	⊃	205	05
f	F	266	66	5	∨	206	06
g	G	67	67	6	∧	07	07
h	H	70	70	7	<	10	10
i	I	271	71	8	>	211	11
j	J	241	41	9	↑	57	57
k	K	242	42	(	[	255	55
l	L	43	43	)	]	256	56
m	M	244	44		!	54	54
n	N	45	45	—	+	40	40
o	O	46	46	•		233	33
p	P	247	47	,	=	73	73
q	Q	250	50	.	×	221	21
r	R	51	51	/	?	272	72
s	S	222	22	Lower Case		274	74
t	T	23	23	Upper Case		200	00
u	U	224	24	Space		75	75
v	V	25	25	Bk. Sp.		236	36
w	W	26	26	Tab		277	77
x	X	227	27	Carr. Ret.		00	00
y	Y	230	30	Tape Feed		--	35
z	Z	31	31	Red*		--	34
				Blk*		13	--
				Stop Code		100	--
				Delete			

*Used on type-out only, not on keyboard.